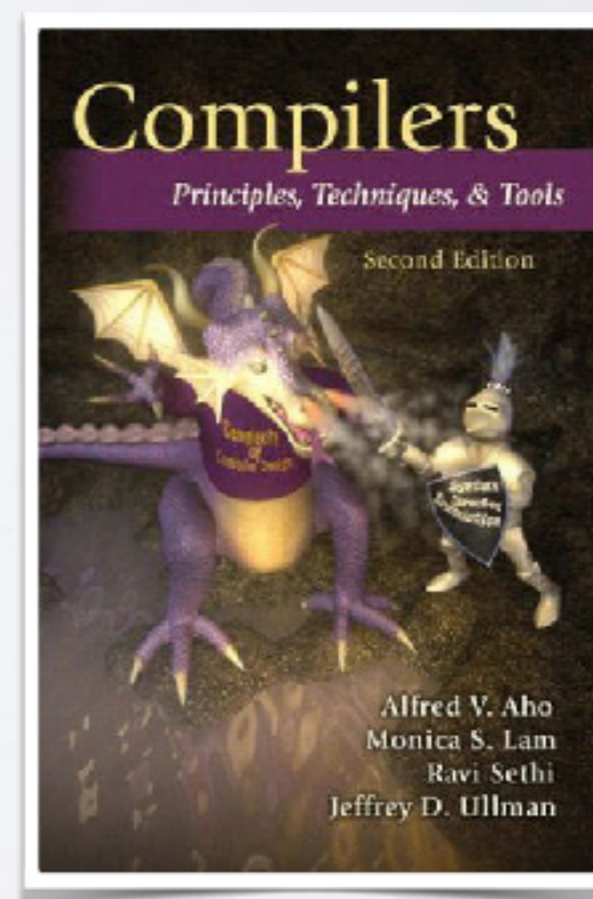
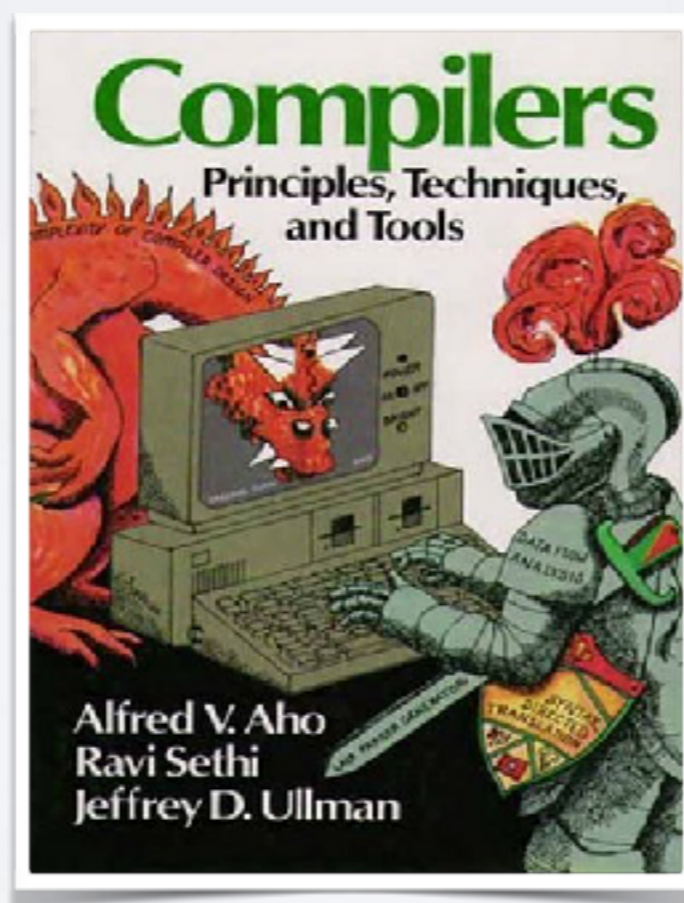


ОСНОВЫ ПРОГРАММНОГО КОНСТРУИРОВАНИЯ

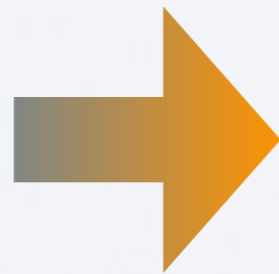
Лекция № 12
21 ноября 2016 г.



ТРАНСЛЯТОРЫ

Компиляторы, интерпретаторы, препроцессоры, конверторы,
парсеры, анализаторы, форматтеры, проверяторы, ...

Текст



Нечто полезное

КОМПИЛЯТОР

Исходный
текст



Анализ

Промежуточное
представление

часто:
синтаксическое дерево



Синтез

Целевая
программа

ОБ АНАЛИЗЕ

- Этапы:
 - Лексический анализ (*лексика* = «слова»).
 - Синтаксический анализ (*синтаксис* = *структура*).
 - Семантический анализ (*семантика* = *смысл*).
- Не во всех задачах трансляции присутствуют все этапы!

ЛЕКСИЧЕСКИЙ АНАЛИЗ (LEXER / SCANNER)

Пример: `a = qq * (c--)`

1. Идентификатор `a`.
2. Знак присвоения.
3. Идентификатор `qq`.
4. Знак умножения.
5. Левая скобка.
6. Идентификатор `c`.
7. Оператор декремента.
8. Правая скобка.

ЛЕКСИЧЕСКИЙ АНАЛИЗ (LEXER / SCANNER)

- На входе: текст.
- На выходе: поток лексем.
- «Съедаются» пробелы, переносы строки, часто – комментарии.
- Лексемы бывают:
 - Без атрибутов (означающие сами себя).
 - С атрибутом («идентификатор такой-то»). **Лексема = Токен + Атрибут.**
- Один и тот же поток лексем может порождаться различными текстами на входе.
- Ошибка: «не могу сделать лексему из того, что дали».

ИНТЕРФЕЙС СКАНЕРА

```
enum Token {
    LP = '(', RP = ')', ADD = '+', SUB = '-',
    MUL = '*', DIV = '/', NUM = 256, EOS, NONE
};

/* Remove current and get next. */
int lex_next();

/* Get current. */
int lex_lookahead();

/* Attribute of current. */
int lex_value();
```

РУЧНАЯ РЕАЛИЗАЦИЯ СКАНЕРА

```
static int number, token = NONE;
int lex_next() {
    for (;;) {
        int c = getchar();

        if (c == EOF) {
            return token = EOS;
        } else if (isspace(c)) {
            continue;
        } else if (isdigit(c)) {
            number = c - '0';
            while ((c = getchar()), isdigit(c))
                number = number * 10 + c - '0';

            ungetc(c, stdin);
            return token = NUM;
        } else {
            return token = c;
        }
    }
}

int lex_value() { return number; }
int lex_lookahead() { return token == NONE ? lex_next() : token ;}
```


ПОСТРОЕНИЕ СКАНЕРОВ

- Недетерминированный конечный автомат (НКА):
 - Набор состояний.
 - Стартовое состояние.
 - Алфавит.
 - Переходы:
Состояние \rightarrow { Набор возможных состояний }.
 - Набор финальных состояний.

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

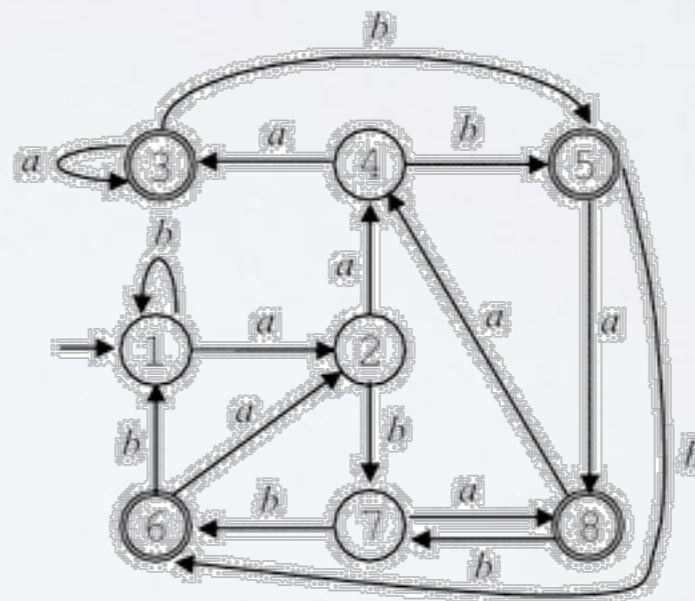
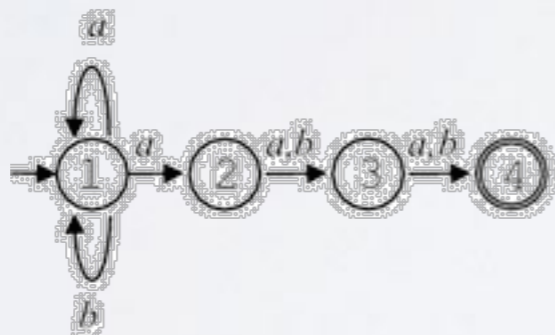
Символ	Значение	Символ	Значение
a	Символ a	$a?$	$(a \epsilon)$
$a b c$	Либо a , либо b , либо c	a^+	aa^*
a^*	Повторение a – 0 или более раз	$[a-z]$	$a b \dots z$
(r)	Подвыражение	$.$	Любой символ

Пример выражения: $(a|b)^*a(a|b)(a|b)$

НКА И ДКА

Детерминированный конечный автомат: из каждого состояния переход в следующее состояние однозначен.

НКА



ДКА

Конечные автоматы для $(a|b)^*a(a|b)(a|b)$

ПОСТРОЕНИЕ СКАНЕРА

- Набор соответствий «Рег. выражение – токен».
- Итог: один большой автомат с финальными состояниями там, где кончаются отдельные токены.

Рег. выражение	Токен
(LP
)	RP
+	ADD
-	SUB
*	MUL
/	DIV
[0-9]+	NUMBER

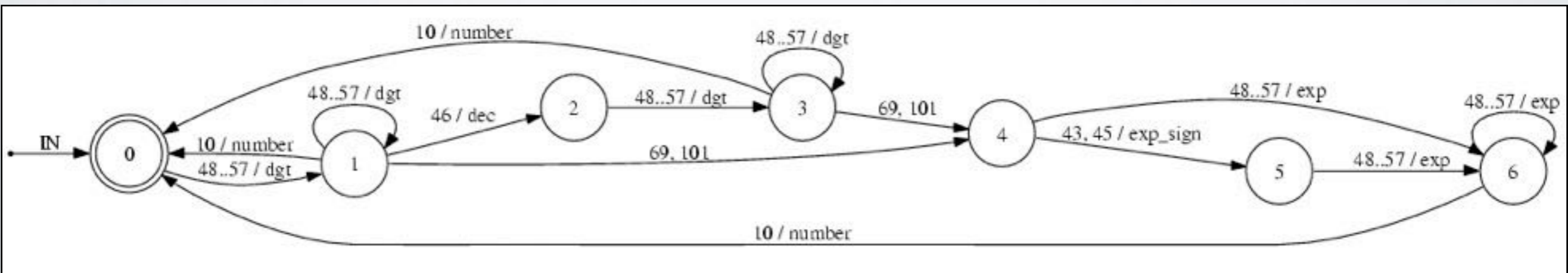
ГЕНЕРАТОР RAGEL

```
action dgt      { printf("DGT: %c\n", fc); }
action dec      { printf("DEC: .\n"); }
action exp      { printf("EXP: %c\n", fc); }
action exp_sign { printf("SGN: %c\n", fc); }
action number   { /*NUMBER*/ }

number = (
  [0-9]+ $dgt ( '.' @dec [0-9]+ $dgt )?
  ( [eE] ( [+\\-] $exp_sign )? [0-9]+ $exp )?
) %number;

main := ( number '\\n' )*;
```

```
st0:
  if ( ++p == pe )
    goto out0;
  if ( 48 <= (*p) && (*p) <= 57 )
    goto tr0;
  goto st_err;
tr0:
  { printf("DGT: %c\n", (*p)); }
st1:
  if ( ++p == pe )
    goto out1;
  switch ( (*p) ) {
  case 10: goto tr5;
  case 46: goto tr7;
  case 69: goto st4;
  case 101: goto st4;
  }
  if ( 48 <= (*p) && (*p) <= 57 )
    goto tr0;
  goto st_err;
```



СИНТАКСИЧЕСКИЙ АНАЛИЗ

- Понятие порождающей грамматики: $G = (\Sigma, N, S, P)$.
 - Σ – алфавит терминальных символов, или *терминалов* (тех символов, которые можно использовать для записи предложений языка).
 - N – алфавит нетерминальных символов (*нетерминалов*, вспомогательных символов, металингвистических или синтаксических переменных).
 - S – символ из N (начальный/стартовый символ).
 - P – набор правил вывода. Формат правила: $\alpha \rightarrow \beta$, где α и β – цепочки, состоящие из символов ($\Sigma \cup N$).

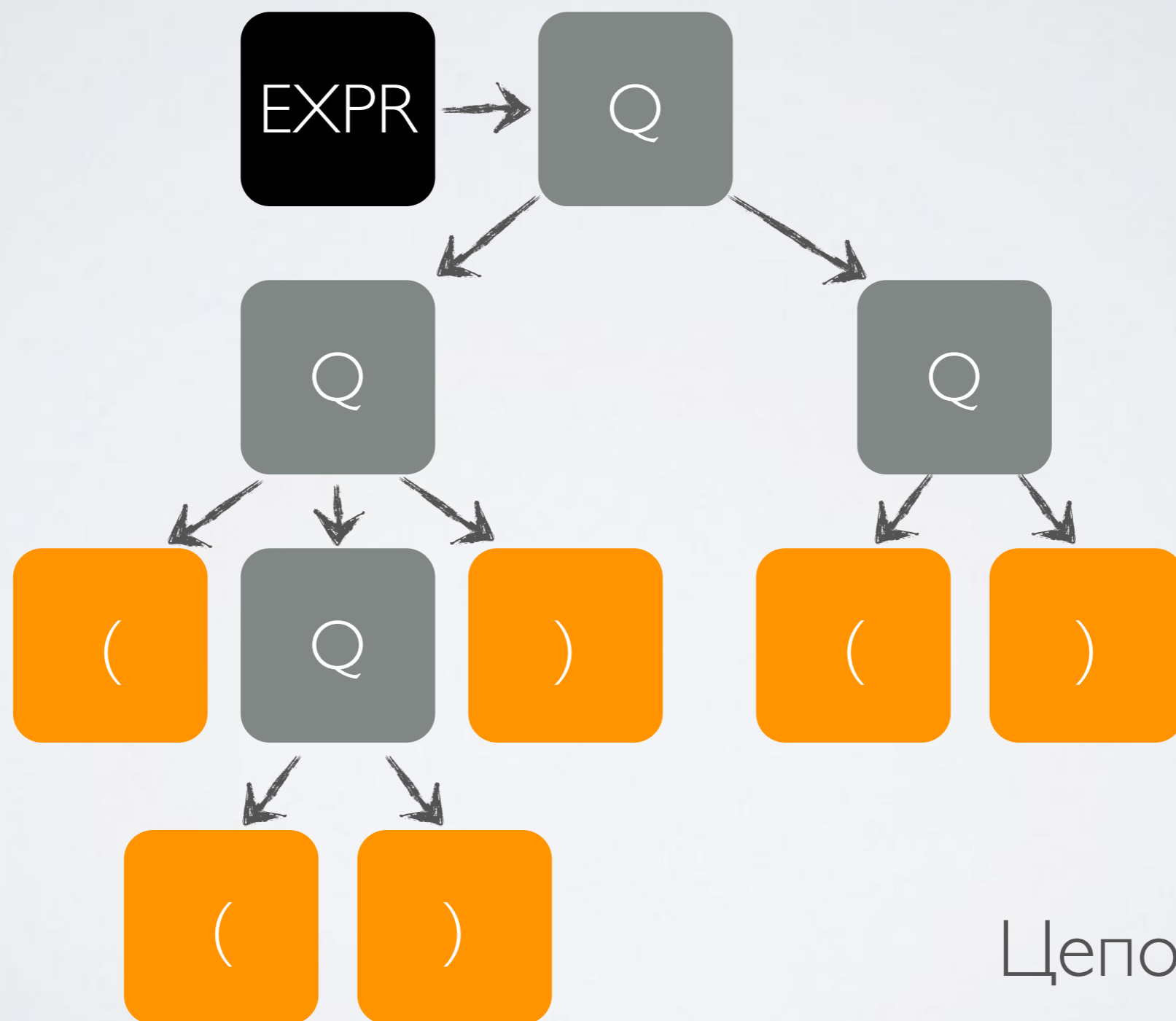
СКОБОЧНАЯ ГРАММАТИКА

- Терминалы (Σ): '(' и ')'.
 - Правила вывода (P):
 - $EXPR \rightarrow Q$
 - $Q \rightarrow ()$
 - $Q \rightarrow (Q)$
 - $Q \rightarrow QQ$
- Нетерминалы (N):
 $EXPR$ и Q .
- Стартовый нетерминал (S):
 $EXPR$.

Корректные выражения: $()$, $()()()$, $(((((0))))0)()$, ...

Некорректные выражения: $)()$, $(((($, (aa)

ДЕРЕВО РАЗБОРА



КОНТЕКСТНО-СВОБОДНЫЕ ГРАММАТИКИ

- Ограничение на правила вывода: только $N \rightarrow \beta$ (N – нетерминал).
- Свойство *локальности* и *независимости* подстановок: подстановка, выполняемая в каком-то месте, не влияет на прочие подстановки за пределами этого места.
- Достаточно полно описывают существующие языки программирования.
- Есть алгоритмы синтаксического анализа.

ГРАММАТИКА ВЫРАЖЕНИЙ

- Наивный подход:

- $E \rightarrow E + E$

- $E \rightarrow E - E$

- $E \rightarrow \text{NUM}$

- $2-3-4$: $(2-3)-4$ или $2-(3-4)$?

- Левоассоциативные и правоассоциативные операторы.



УЧИТЫВАЕМ АССОЦИАТИВНОСТЬ

- Шаг в нужную сторону:
 - $E \rightarrow E + \text{NUM}$
 - $E \rightarrow E - \text{NUM}$
 - $E \rightarrow \text{NUM}$
- Более короткая запись:
 - $E \rightarrow E + \text{NUM} \mid E - \text{NUM} \mid \text{NUM}$

ДОБАВЛЯЕМ УМНОЖЕНИЕ

Так пойдет?

$$E \rightarrow E + NUM \mid E - NUM \mid E * NUM \mid E / NUM \mid NUM$$

Не учтены приоритеты!

В дереве разбора операции * и /
должны стоять пониже, ближе к листьям.

ПРАВИЛЬНОЕ УМНОЖЕНИЕ

- Добавляем еще один нетерминал, **F**:

- $E \rightarrow E + T \mid E - T \mid T$

- $T \rightarrow T * F \mid T / F \mid F$

- $F \rightarrow \text{NUM}$

- А как добавить скобки?

- $F \rightarrow (E)$

НАПРАВЛЕНИЯ РАЗБОРА

- Нисходящий разбор (от корня к листьям).
- Восходящий разбор (от листьев к корню).

МЕТОД РЕКУРСИВНОГО СПУСКА

- Нисходящий разбор.
- Годится только для грамматик класса **LL(1)**:
 - Отсутствие левой рекурсии ($A \rightarrow A\beta$).
 - Однозначность.
 - Глубина просмотра 1 (текущий токен однозначно определяет правило вывода).

ГРАММАТИКА ВЫРАЖЕНИЙ И РЕКУРСИВНЫЙ СПУСК

- Годная грамматика:

- $E \rightarrow T \{ [+ -] T \}^*$

- $T \rightarrow F \{ [* /] F \}^*$

- $F \rightarrow \text{NUM} \mid (E)$

- На каждый нетерминал заводим отдельную функцию.

ИСХОДНЫЙ КОД ПРОВЕРЯТОРА

```
void E() {  
    T();  
    for (;;) {  
        if (lex_lookahead() == ADD || lex_lookahead() == SUB) {  
            lex_next();  
            T();  
        } else {  
            break;  
        }  
    }  
}
```

```
void F() {  
    if (lex_lookahead() == LP) {  
        lex_next();  
        E();  
        match(RP);  
    } else {  
        NUM();  
    }  
}
```

```
void match(int token) {  
    if (lex_lookahead() == token) {  
        lex_next();  
    } else {  
        error();  
    }  
}
```

$$E \rightarrow T \{ [+ -] T \}^*$$
$$T \rightarrow F \{ [* /] F \}^*$$
$$F \rightarrow \text{NUM} \mid (E)$$

ПОЛЕЗНАЯ ПРОДУКЦИЯ

- При вычислении выражения:
 - Каждая функция возвращает значение подвыражения.
- При построении синтаксического дерева:
 - Каждая функция возвращает указатель на построенный ей узел дерева.
 - Потомками этого узла являются значения, возвращенные другими функциями-нетерминалами.

ОБРАБОТКА ОШИБОК

- Функция `match()`.
- В приведенном примере все нераспознанное попадает в функцию `NUM()`.
- Функция, соответствующая стартовому нетерминалу, должна «проглотить» все выражение.

ПОЛНЫЙ ПРИМЕР

Исходный код лексера и парсера для вычисления простых арифметических выражений:

https://github.com/cypok/simple_parser

Tokenization of $\langle 2 + 2 * 2 + (37 - 35) \rangle$:
2 ADD 2 MUL 2 ADD LP 37 SUB 35 RP EOS

Evaluated $2 + 2 * 2 + (37 - 35)$ to 8



© Scott Adams, Inc./Dist. by UFS, Inc.

КОНЕЦ ДВЕНАДЦАТОЙ ЛЕКЦИИ